A Novel Bit-Wise Adaptable Entropy Coding Technique[1]

Aaron Kiely and Matthew Klimesh
Jet Propulsion Laboratory, California Institute of Technology
4800 Oak Grove Drive, Mail Stop 238-420, Pasadena, CA 91109
e-mail: {aaron, klimesh}@shannon.jpl.nasa.gov

Abstract— We present a novel entropy coding technique which is based on recursive interleaving of variable-to-variable length binary source codes. The encoding is adaptable in that each bit to be encoded may have an associated probability estimate which depends on previously encoded bits. The technique can achieve arbitrarily small redundancy. The technique may have advantages over arithmetic coding, including most notably a simple and fast decoder.

1 Outline of The Entropy Coding Technique

1.1 Introduction

In data compression algorithms the need frequently arises to compress a binary sequence in which each bit has some estimated distribution, i.e., probability of being equal to zero. In many practical situations it is desirable to have the estimated distribution for a bit depend on the values of earlier bits. Accommodating such a dynamically changing probability estimate is tricky because the decoder must make the same estimate as the encoder. Thus, before the ith bit can be decoded, the value of the first $i-1$ bits must be determined.

To our knowledge, currently the only efficient encoding methods in this case are arithmetic coding and the relatively unknown technique called interleaving entropy coders [1], which is a generalization of the "block Melcode" [2]. In this paper, we describe a new entropy coding technique which can efficiently encode a binary source with a bit-wise adaptive probability estimate. This technique can be seen as a generalization of the interleaving entropy coders method [1].

1.2 The Source Coding Problem

We examine the problem of compressing a sequence of bits $b_1, b_2, \ldots$ from a random source. The source probability estimates $p_i = \text{Prob}[b_i = 0]$ may depend on the values of the source sequence prior to index i. In general, this dependence encompasses both adaptive probability estimation as well as correlations or memory in the source. Consequently, efficient encoding requires a bit-wise adaptable encoder.

We make the following assumptions:

[1]The work described was funded by the TMOD Technology Program and performed at the Jet Propulsion Laboratory, California Institute of Technology under contract with the National Aeronautics and Space Administration.

1. Without loss of generality, we assume that $p_i \geq 1/2$ for each index i. If this were not the case, we could simply invert bit b_i before encoding to make it so.

2. We also assume that the decoder can determine when decoding is complete. In practice, this often occurs automatically, or straightforward methods can be used, such as transmitting the sequence length prior to the compressed sequence.

1.3 The Recursive Interleaved Entropy Coder Concept

Since, by assumption, each bit has probability of zero at least 1/2, we are concerned with the probability region $[1/2, 1]$. We partition this region into several narrow intervals, and with each interval we associate a bin that will be used to store bits. When bit b_i arrives, we place it into the bin corresponding to the interval containing p_i. Because each interval spans a small probability range, all of the bits in a given bin have nearly the same probability of being zero, and we can think of each bin as corresponding to some nominal probability value.

For each bin (except the leftmost bin, which contains probability 1/2) we specify an exhaustive prefix-free set of binary codewords. When the bits collected in a bin form one of these codewords, we delete these bits from the bin and encode the value of the codeword by placing one or more new bits in other bins[2]. This process is conveniently described using a binary tree. Each codeword is assigned to a terminal node in the tree, internal nodes are labeled with a destination bin, and the branch labels (each a zero or one) correspond to the output bits that are placed in the destination bins.

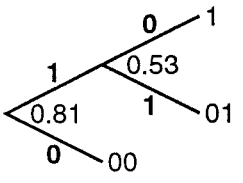

Figure 1: Example of a tree for a bin with representative probability 0.9.

For example, Figure 1 shows a tree that might be used for a bin with nominal probability 0.9. The prefix-free codeword set for this bin is $\{00, 01, 1\}$, shown as labels of the terminal nodes in the tree. If the codeword to be processed in the bin is 00, which occurs with probability approximately 0.81, we place a zero in the bin that contains probability 0.81. If the codeword is 1, first we place a one in the bin containing probability 0.81, which indicates that the codeword is something other than 00, then we place a zero in the bin containing probability 0.53 because, given that the codeword is not 00, the conditional probability that the codeword is 1 is approximately 0.53.

For the first bin we do not define a tree such as the one in Figure 1. Instead, bits in this bin form the encoder's output. Bits that reach the first bin have probability of being zero very close

[2]The ordering of the new bits in a bin is not straightforward and we save these details for Section 2.2.

to 1/2 and are thus nearly incompressible, so transmitting these bits uncoded does not add much redundancy.

Bits arrive in various bins either directly from the source or as a result of processing codewords in other bins. Our goal is to have new bits migrate to the leftmost (uncoded) bin, where they are transmitted. To accomplish this, we require trees to be designed so that all new bits resulting from the processing of each codeword are placed in bins strictly to the left of the bin in which the codeword was formed. Apart from our desire to move bits to the left, this requirement also prevents encoded information from traveling in "loops", which would make coding difficult or impossible. Thus if a bin has nominal probability p, we would like the probability of a zero for each output bit to be in the range $[1/2, p)$. Perhaps surprisingly, this turns out to be a reasonable requirement:

Theorem 1 *For any given probability value $p \in (1/2, 1)$, there exists a tree with the property that all output bits have probability of zero in the range $[1/2, p)$. Such a tree is said to be* **useful** *at p.*

This is proven by exhibiting an infinite family of trees for which at least one tree is useful at any $p \in (1/2, 1)$. Figure 2 illustrates a tree from this family. We omit the details of the proof.

When we reach the end of the bit sequence to be encoded and no codewords remain in any bin, there will generally be partially formed codewords in one or more bins. Since these bits are needed for decoding, we append one or more extra bits to each of these partial codewords to form complete codewords which are then processed in the normal manner.

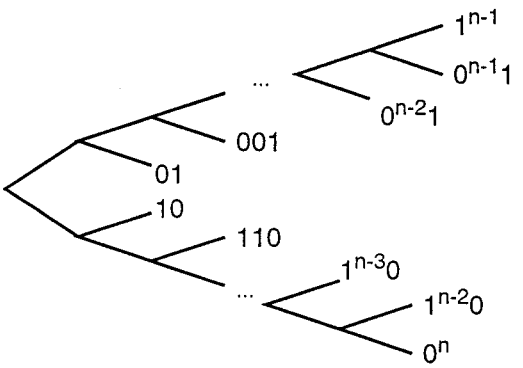

Figure 2: A useful tree.

In practice, the encoder and decoder do not keep track of probability values. Instead, each bin is assigned an index, starting with index 1 corresponding to the leftmost (uncoded) bin. At each node in the tree we identify the index (rather than the nominal probability value) of the bin to which the next output bit is mapped. The requirement we impose is that each output bit from the tree for bin j must be mapped to a bin with index strictly less than j. No computations involving probability values are needed except that at the input it will be necessary to map each bit b_i to the appropriate bin index[3].

We can see intuitively that some redundancy is present in this system because the bins have positive width — the probability associated with a bit that arrives in a bin will usually not exactly

[3] In fact, we don't necessarily need to know (or estimate) p_i if we have another good method for assigning the appropriate bin index to each input bit.

equal the bin's nominal probability, and bits in the leftmost bin are transmitted uncompressed even though they may not have probability of zero exactly equal to 1/2. As one might expect, however, by increasing the number of bins and/or the size of the trees, we can trade complexity for performance and decrease the maximum redundancy to arbitrarily small values.

An important special case of the entropy coder arises when all output bits generated from each tree in the encoder are mapped to the uncoded bin. In this case the encoder amounts to interleaving several separate variable-to-variable length binary codes, as in [1, 2].

2 Encoding and Decoding

2.1 Decoder Operation

We first describe decoding since it determines the encoding procedure. It is convenient to think of each bin in the decoder as containing a list of bits. To decode, we initially place all of the encoded bits in the first (uncoded) bin, and all other bins are empty. At any time, each nonempty bin (with the exception of the uncoded bin) will contain a single codeword or a prefix of a codeword.

Software decoding uses two recursive procedures, `GetBit` and `GetCodeword`. `GetBit` simply takes the next available bit from the indicated bin. If the bin is empty then it first calls `GetCodeword`. Given an empty bin, `GetCodeword` determines which codeword must have occupied the bin by taking bits from other bins (via `GetBit`), then places that codeword in the bin. The `GetCodeword` procedure is similar to Huffman decoding, except that at each step we take the next bit from the appropriate bin, not (necessarily) from the encoded bit stream.

To decode the ith bit, let `binindex` equal the index of the bin to which the ith bit would have been assigned. This assignment may be a function of all of the previously decoded bits. Then the ith decoded bit is equal to `GetBit(binindex)`.

2.2 Encoder Operation

To ensure that decoding is possible, we must pay careful attention to the order in which bits are processed by the encoder. The bin to which bit b_i is mapped may depend on the values of previous bits, and therefore information needed to encode bit b_{i-1} has priority over (i.e., should be processed before) information needed to encode bit b_i.

One encoding method that produces encoded bits in the appropriate order is to maintain a linked list of bit values sorted in order of priority. Each record in the list stores the bit value and

the index of the bin that contains the bit. We can start by placing the entire input sequence in the list. To encode, at each step we start with the highest indexed nonempty bin. We take bits from this bin (in priority order) until we have formed a codeword, appending flush bits if needed. We delete the bits that formed the codeword and insert the resulting output bits in the list at the location of the highest priority bit in the codeword.

To manage long input sequences with limited memory, we can partition the input sequence into blocks of known size and encode each block separately. An alternative technique is not described here due to space limitations.

3 Code Design

In this section we'll illustrate a procedure to design an encoder. To apply this procedure we begin with a redundancy target Δ which is the maximum allowed redundancy[4] (in bits) and a set of candidate trees to be used in the encoder. In this context each tree does not include assignments of bin indices to internal nodes or output bit labels to branches. These assignments will be made as part of the design procedure. With a suitable set of candidate trees, we can produce an encoder that has redundancy less than Δ for arbitrarily small Δ.

We select the tree for each bin in order of increasing bin index. When bins $1, 2, \ldots, j-1$ have been completed, designing the jth bin amounts to selecting a tree for the bin, assigning bin indices to internal nodes and output bit labels to branches, and calculating z_{j-1}, the probability value where we switch from bin $j-1$ to bin j. (Of course no design work is required for the first bin since it is uncoded and $z_0 = 1/2$.) For example, Figure 3 shows a case where the encoder has been designed for the first three bins, and our redundancy target Δ is met when p is less than some critical value p^*. Thus we know that $z_3 \leq p^*$, and we need to specify the tree to use for the fourth bin of the encoder.

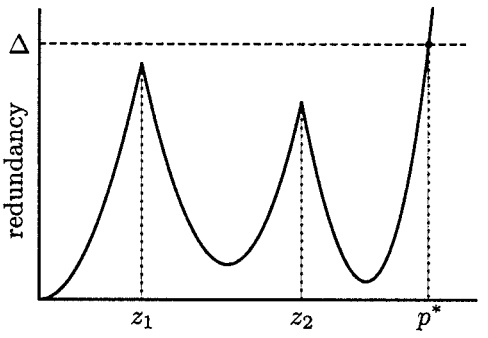

Figure 3: Redundancy of an encoder after designing the first three bins.

To do this, we can take from our set of candidate trees any tree that is useful at p^* and assign branch and internal node labels based on this probability value. That is, we calculate the branch

[4]We assume here that the maximum redundancy will occur at the boundary between two bins.

probability for each internal node in the tree and label the branches so that a zero output is more likely than a one at each node. Then, at each internal node, if a zero output bit occurs with probability q, we map this bit to the bin with index j such that $q \in [z_{j-1}, z_j)$.

This construction maps output bits to bins in regions where the redundancy is less than the target Δ, and it can be shown that the redundancy at probability p^* is strictly less than Δ. Since the rate functions for each bin are continuous, we have extended the range where the encoder meets the redundancy bound.

We can also try assigning branch labels, and even selecting a tree, based on some probability target value larger than p^*. This alternative generally produces larger redundancy at p^*, but frequently meets the redundancy target Δ at p^* and may extend further the range over which bin j is used, which can help to reduce the total number of bins used in the encoder.

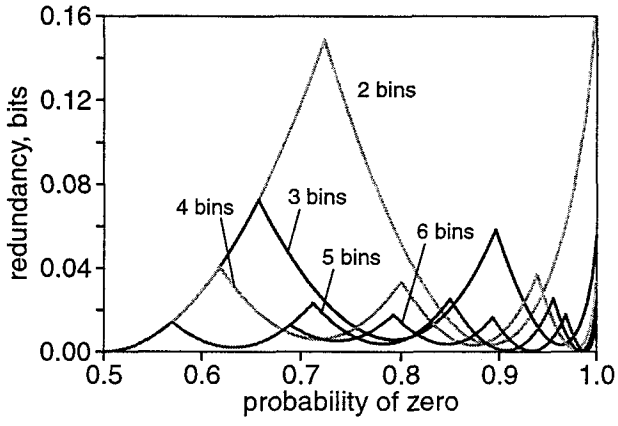

Figure 4: Redundancy of some encoders using a small number of bins.

We have found some good encoder designs using this technique. Figure 4 shows the redundancy of some of these designs, computed using a recursive rate estimation technique we developed.

4 Conclusion

The techniques described here have been used to produce working software encoders and decoders that confirm the performance estimates shown. Low redundancy is attainable using relatively small trees (e.g., an average of 6 terminal nodes in the trees used for the encoders of Figure 4). This technique may be a viable alternative to arithmetic coding when decoding speed is important.

References

[1] P. G. Howard, "Interleaving Entropy Codes," *Proc. Compression and Complexity of Sequences 1997* pp. 45–55, 1998.

[2] F. Ono, S. Kino, M. Yoshida, and T. Kimura, "Bi-Level Image Coding with MELCODE — Comparison of Block Type Code and Arithmetic Type Code," *Proc. IEEE Global Telecommunications Conference (GLOBECOM '89)*, pp. 0255 - 0260, Nov. 1989.